Language Basics

Reading:

- Java Tutorial: Section *Language Basics* under section *Learning the Java Language*
- Textbook: Section 1.2 & 1.4 (1.3 will be covered next week)

**NOTE (VERY IMPORTANT):** These materials are **NOT** the substitutions of original the texts. You are strongly recommended to study the articles mentioned above to get all necessary details.

**A quote from your textbook:**

*Once a person has understood the way variables are used in programming, he has understood the quintessence of programming.*

E. W. DIJKSTRA, *Notes on Structured Programming*
Variables:

As you learned in the previous lesson, an object stores its state in fields

```java
int cadence = 0;
int speed = 0;
int gear = 1;
```

Java defines the following kind of variables:

- **Instance variables (non-static fields)**
  - Objects store their individual states in "non-static fields", that is, fields declared without the static keyword.
  - Non-static fields are also known as instance variables because their values are unique to each instance of a class
  - The `currentSpeed` of one bicycle is independent from the `currentSpeed` of another

- **Class variables (static fields)**
  - A *class variable* is any field declared with the static modifier
  - This tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated
- A field defining the *number of gears* for a particular kind of bicycle could be marked as static since conceptually the same number of gears will apply to all instances.
- The code `static int numGears = 6;` would create such a static field.
- Additionally, the keyword `final` could be added to indicate that the number of gears will never change `final static int numGears = 6`.

**Local variables**
- Similar to how an object stores its state in fields, a method will often store its temporary state in local variables.
- Syntax is similar to declaring a field.
  - Example: `int count = 0;`
- Declared in between the opening and closing braces of a method.
- Local variables are only visible to the methods in which they are declared.

**Parameters**
- Recall that the signature for the main method is `public static void main(String[] args)`.
  - Here, the args variable is the parameter to this method.
- Parameters are always classified as "variables" not "fields". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers).
Scope:

- A variable's scope is the region of a program within which the variable can be referred to by its simple name
- Scope also determines when the system creates and destroys memory for the variable

The location of the variable declaration within your program establishes its scope. There are four categories of scope, as shown in the following figure:
Naming (rules and conventions):

- Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "$", or the underscore character "_"
- The convention, however, is to always begin your variable names with a letter, not "$" or "_". Additionally, the dollar sign character, by convention, is never used at all
- Subsequent characters may be letters, digits, dollar signs, or underscore characters.
- When choosing a name for your variables, use full words instead of cryptic abbreviations.
- The name you choose must not be a keyword or reserved word

Variable Naming Convention

- If the name you choose consists of only one word, spell that word in all lowercase letters.
- If it consists of more than one word, capitalize the first letter of each subsequent word.
- The names gearRatio and currentGear are prime examples of this convention.
- If your variable stores a constant value, such as static final int NUM_GEAR = 6, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character.
- By convention, the underscore character is never used elsewhere.
## Primitive Data Types:

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>KIND OF VALUE</th>
<th>MEMORY USED</th>
<th>SIZE RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true or false</td>
<td>1 byte</td>
<td>Not applicable</td>
</tr>
<tr>
<td>char</td>
<td>Single character (Unicode)</td>
<td>2 bytes</td>
<td>Common Unicode characters</td>
</tr>
<tr>
<td>byte</td>
<td>Integer</td>
<td>1 byte</td>
<td>−128 to 127</td>
</tr>
<tr>
<td>short</td>
<td>Integer</td>
<td>2 bytes</td>
<td>−32768 to 32767</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
<td>4 bytes</td>
<td>−2147483648 to 2147483647</td>
</tr>
<tr>
<td>long</td>
<td>Integer</td>
<td>8 bytes</td>
<td>−29223372036854775808 to 29223372036854775807</td>
</tr>
<tr>
<td>float</td>
<td>Floating-point number</td>
<td>4 bytes</td>
<td>±3.40282347 × 10^{+38} to ±1.40239846 × 10^{-45}</td>
</tr>
<tr>
<td>double</td>
<td>Floating-point number</td>
<td>8 bytes</td>
<td>±1.76769313486231570 × 10^{+308} to ±4.94065645841246544 × 10^{-324}</td>
</tr>
</tbody>
</table>

Display 1.2 from textbook
Default values:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default Value (for fields)</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>0L</td>
</tr>
<tr>
<td>float</td>
<td>0.0f</td>
</tr>
<tr>
<td>double</td>
<td>0.0d</td>
</tr>
<tr>
<td>char</td>
<td>'\u0000'</td>
</tr>
<tr>
<td>String (or any object)</td>
<td>null</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
</tbody>
</table>

Literals:

- Primitive types are special data types; they are not objects created from a class
- A literal is the source code representation of a fixed value
- it's possible to assign a literal to a variable of a primitive type

    boolean result = true;
    char capitalC = 'C';
    byte b = 100;
    short s = 10000;
    int i = 100000;
## Examples of Literal Values

<table>
<thead>
<tr>
<th>Literal Value</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>Int</td>
</tr>
<tr>
<td>8864L</td>
<td>Long</td>
</tr>
<tr>
<td>37.266</td>
<td>Double</td>
</tr>
<tr>
<td>37.266D</td>
<td>Double</td>
</tr>
<tr>
<td>87.363F</td>
<td>Float</td>
</tr>
<tr>
<td>26.77e3</td>
<td>Double</td>
</tr>
<tr>
<td>'c'</td>
<td>Char</td>
</tr>
<tr>
<td>True</td>
<td>Boolean</td>
</tr>
<tr>
<td>False</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

### Integer Literals:

// The number 26, in decimal  
int decVal = 26;
// The number 26, in hexadecimal  
int hexVal = 0x1a;
// The number 26, in binary  
int binVal = 0b11010;
Floating-point Literals:

double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;

Character and String Literals

- "Unicode escape" such as '\u0108' (capital C with circumflex), or "S\u00ED Se\u00F1or" (Sí Señor in Spanish)
- Always use 'single quotes' for char literals and "double quotes" for String literals
- The Java programming language also supports a few special escape sequences for char and String literals:

\b (backspace)
\t (tab)
\n (line feed)
\f (form feed)
\r (carriage return)
\" (double quote)
\\ (single quote)
\\ (backslash)
There's also a special `null` literal that can be used as a value for any reference type. `null` is often used in programs as a marker to indicate that some object is unavailable.

Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending ".class"; for example, `String.class`. This refers to the object (of type `Class`) that represents the type itself.

**Using Underscore Characters in Numeric Literals**

```java
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCafe_Babe;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;
// OK (decimal literal)
int x3 = 5_______2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;
/ OK (hexadecimal literal)
int x6 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;

**Final variables**
The value of a final variable cannot change after it has been initialized. Such variables are similar to constants in other programming languages.

final int aFinalVar = 0;